CONSTANT AND READ ONLY

If we declare something as constant we cant change it in run time

Const I = 2147483647

Const j

Unchecked {

Int Z = I + j

}

Types of conversion  
c++,

parse,

Convert classes

Boxingg

It doesn’t accept null values

Int I = null; {wrong}

Int? I = null; {right}

Nullable<int> I = Null;

Parse and TryParse

Static void main(string[] args)

{

Int I =0;

Bool b = int.TryParse(“10z”,out i);

Console.WriteLine(i);

Console.ReadLine();

**CONVERSION**

A ***conversion*** causes an expression to be converted to, or treated as being of, a particular type; in the former case a conversion may involve a change in representation. Conversions can be ***implicit*** or ***explicit***, and this determines whether an explicit cast is required.These are few of the imp implicit conversions

**IMPLICIT CONVERSIONS**

* **Implicit nullable**

Int I = null; {wrong}

Int I =42

Int? j = i ; {right}

**int?** is a nullable value type, and **I**  is a regular **int**. The assignment **int? j = i;** is an implicit nullable conversion because the compiler understands that it can safely assign a non-nullable value to a nullable version of the same type.

* **Implicit numeric**

From **sbyte** to **short, int, long, float, double, or decimal.** It is like converting one datatype to another data without data loss

* **Implicit reference conversions**
* **Boxing conversions**
* **Implicit constant conversions**

In this case, the implicit constant conversion would happen when adding **float\_number** and **integer\_number**. The integer constant **5** would be implicitly converted to a floating-point value before the addition operation is performed.

**PARSE AND TRY PARSE**

You convert the string to numbers using parse or try parse method.

Try parse example

Bool b = Int.TryParse(“vysh”,out int i);

Parse Example

Int I = int.Parse(“hello”);

The Parse method returns the converted number;

* The **TryParse method returns a boolean** value that indicates whether the conversion succeeded, and returns the converted number in an out parameter.
* If the string isn't in a valid format, Parse throws an exception, but TryParse returns false.
* When calling a Parse method, you should always use exception handling to catch a **[FormatException](https://learn.microsoft.com/en-us/dotnet/api/system.formatexception)** when the parse operation fails.

**EXPLICIT CONVERSION**

For explicit conversion we either use type casting or Convert class in c# .

If we use type casting no matter what datatype we specify or throw we don’t get exception and it successfully coverts into the other datatype .But convert class throw an exception when it cannot handle the values i.e the overflow exception will occur

Type casting eg

Float f;

Int I = (int)f

Convert class in c#

Float f = 10.35f

Int I = Convert.Int32(f)

**EXTENSION METHODS IN C#**

In C#, an extension method is a special kind of static method that allows you to add new methods to existing types without modifying them. Extension methods are a syntactic convenience and are used to enhance the functionality of types, even those for which you don't have access to the source code.

**In the c# we have SOLID design principle in which O represents the open closed principle. That means open for extension and closed for modification.**

That is why when we face a situation where we should add the additional functionality we then will not modify the existing code that is already in the production but extend the code in some other file

**IENUMERABLE vs IENUMERATOR**

**IEnumerable** is the interface that allows a collection to be enumerated, and it provides an **IEnumerator** for this purpose. **IEnumerator**, on the other hand, is responsible for maintaining the state of the iteration and providing access to the current element. Understanding these interfaces is crucial when working with custom collections or implementing custom iterators.

* **IEnumerable** doesn't keep track of the iteration state; it delegates this responsibility to **IEnumerator**.
* **IEnumerator** maintains the state information (current position) during the iteration.

It is not the IEnumerable or IEnumerator maintains the state or not, but the technique of iteration through IEnumerable or IEnumerator rests the position or not. As you loop through IEnumerable with foreach loop, the foreach loop automatically resets the IEnumerable position to the initial position each time you iterate. While as you iterate through IEnumerator with MoveNext() method the position is maintained with the new iteration loop, because nothing automatically changes the position of IEnumerator and if you want the initial position of IEnumerator you must use Reset() method. So, the technique used to iterate through IEnumerable rests the position automatically each time you iterate, while The technique used to iterate through IEnumerator begins the iteration from the last IEnumerator position and no automatic reset is carried out in IEnumerator iteration.

Reference for sand castle:

https://www.c-sharpcorner.com/article/create-documentation-with-sandcastle-help-builder/